CSC324 Assignment 2: Continuation Passing Style Transform

In exercise 10, we explored a style of programming called the **Continuation Passing Style (CPS)**. In this style of programming, we explicitly represent the control flow of our program.

Transforming programs in CPS is useful for several reasons:

1. It allows us to emulate delimited continuation in languages without such features.
2. It is actually used by some compilers as intermediate representation, and is *closer to assembly code*: calling a continuation is analogous to an assembly `jump` statement.
3. Programs written in web languages are often written in CPS, where we provide a "callback" to functions that take a long time to run (for example, those that make HTTP requests).

However, as you saw in exercise 10, writing programs in CPS *by hand* is tedious and error-prone. It is much easier to write programs in direct style. Moreover, transforming functions from direct style to CPS is actually quite mechanical. It would be nice to have a program capable of doing what we did by hand in exercise 10.

That's exactly what we will do in this assignment: we will write a program that converts expressions written in direct style to CPS.

As always, make sure to review this handout, the starter code, and the general assignment guidelines before writing any code.

Due date: December 3rd, 2019 before 10:00pm

Starter code

We will be using `Ex10Bork.hs` from execise 10. All new code for this assignments should be written in `CPSBork.hs`.

- `Ex10Bork.hs`
- `CPSBork.hs`

Do not make any changes to `Ex10Bork.hs`. You will not be submitting this file for grading. Instead, we will supply our own.

Bork

We will be working with the **Bork** language from exercise 10. The expressions and values in this language are described in `Ex10Bork.hs`. We will also use the `eval` and `def` functions in this file. To keep it simple, we won't use the CPS interpreter from Exercise 10, and we won't consider shift and reset expressions. CPS transforming shift and reset expressions is possible, but a *lot* more challenging than the algorithm we will be describing.

Warm-up

To warm up and to become familiar with the language structure, let's consider this definition in Bork:

```
facEnv = def [("fac", Lambda ["n"]
                        (If (Equal (Var "n") (Literal $ Num 0))
                            (Literal $ Num 1)
                            (Times (Var "n") (App (Var "fac")
                               [(Plus (Var "n") (Literal $ Num (-1)))])))))]
```

Create an environment `cpsFacEnv` (using `def`) that contains the factorial function `cps_fac` written in CPS. (Hint: do the same thing that you did in Exercise 10, but in Bork instead of Haskell).

Then, create the environments `fibEnv` and `cpsFibEnv` that contains (respectively) functions `fib` and `cps_fib` that computes the `n`-th Fibonacci number using recursion. The `fib` function should be written in direct style, and `cps_fib` in CPS.

Continuation Passing Style Transform

Our goal for this assignment is to complete the helpers to the function `cpsDef`. This function takes as input a list of `(String, Expr)` pairs, where the `String` is the name of a function, and the `Expr` is a Bork expression. The function returns a new set of name to expression pairs, where the expressions are in CPS. For example, `cpsDef` should be able to take:

```
[("fac", Lambda ["n"]
           (If (Equal (Var "n") (Literal $ Num 0))
               (Literal $ Num 1)
               (Times (Var "n") (App (Var "fac")
                 [(Plus (Var "n") (Literal $ Num (-1)))])))))]
```

... and return ...

```
[("cps_fac", Lambda ["cps_n", "k"] ...etc...)]
```

Notice that the variables are renamed, and the `Lambda` term has an extra argument "k". We'll go through the full set of rules that our CPS transformation will follow.

CPS Transformation Rules

Our `cpsDef` will use the helper function `cpsExpr` to perform CPS transformations on a single expression at a time. Your main task is to complete this `cpsDef` function.

The signature of `cpsDef` might look a little strange. The function takes three arguments:

- The expression to be transformed into CPS
- A *prefix string* to be used for creating new variable names (used for function applications)
- A *context function* that expresses work to be done after CPS transforming this expression. This function takes the result of CPS transforming this expression, and returns the result of CPS transforming any parent expressions. The top-level call of `cpsExpr` from `cpsDef` will have an empty context function (identity), but any recursive calls of `cpsExpr` might have a more involved context function.

The best way to illustrate the way these arguments are used is through examples. To do that, we'll go through the different kinds of expression that we might CPS, and describe the recursive structure. Since using Bork syntax in Haskell is a bit cumbersome, all of our examples will use the equivalent Racket syntax.

CPS Transforming a literal value

The CPS transformation of a (top-level) literal value is simply that literal value.

```
Prelude> cpsExpr (Literal $ Num 1) "" id -- "id" is the identify function
Literal $ Num 1
```

However, that literal value could appear in some more complicated context. So, we return the *context function* applied to that literal value. In our simple example above, that *context function* is the identity. As we explain the CPS transformation rules further, we'll see some examples of context functions that are not the identity function.

CPS Transforming a variable

The CPS transformation of a variable is almost as straightforward. The only difference is that we rename variables by prepending the string `cps_`. Note that the helper function `rename` is provided to you, and is also used in `cpsDef`.

```
Prelude> cpsExpr (Var "x") "" id
Var "cps_x"
Prelude> cpsDef [("m", Var "n")]
[("cps_m", Var "cps_n")]
```

CPS Transforming Plus, Times, and Equal

All three of these builtins operations will follow the same structure, so let's use `Plus` as an example. The result of CPS transforming this expression is

```
Prelude> cpsExpr (Plus (Var "x") (Var "y")) "" id
Plus (Var "cps_x") (Var "cps_y")
```

At first, this example seems straightforward. However, for a *general* CPS transformation algorithm, we actually need to do a few things:

1. CPS transform the left argument of the `Plus`: in our case, the `x`
2. *after that is done* and we get the expression `cps_x`, we need to CPS transform the right argument of `Plus`: the `y`
3. *after that is done* and we get the expression `cps_y`, we need to combine the results in a new `Plus`.
4. *after that is done* we need to call the context function with our result.

If this "context function" sounds like a continuation to you, you're absolutely right! Every time we say "*after that is done*", what we're really doing is adding to the context function. In that sense, writing this CPS transformer might feel like you're still working on Exercise 10.

The reason why `Plus`, `Times` and `Equals` requires all these steps is because the left and right subexpressions may be more complex. For example, we could have *function calls* in these subexpressions. This next example will trace through what happens when the left subexpression contains a function call, and might make more sense after reading how to CPS a function call.

```
Prelude> cpsExpr (Plus (App (Var "f") [Var "n"]) (Literal $ Num 3)) "" id -- (+ (f n) 3)
App (Var "cps_f") [Var "cps_n",Lambda ["result"] (Plus (Var "result") (Literal (Num 3)))]
```

Let's trace through what should happen in this example:

1. We first CPS transform the left argument of the Plus, which is a function application. So, there is a recursive call to `cpsExpr` with the expression `(App (Var "f") [Var "n"])`. In this recursive call, the last parameter (context function) will no longer be the identity function `id`. Instead, it will be a function that performs steps 2-4 below:
2. CPS transform the right argument of the Plus, which is `Literal (Num 3)`. So, there is another recursive call to `cpsExpr` with this expression, and the last argument in this recursive call is a function that performs steps 3-4
3. Perform the addition.
4. Apply the context function from all the way back in step 1, which, in this case, is the identify function `id`.

CPS Transforming Lambda

Now we are getting to the more interesting cases. To CPS transform a function definition, we need to:

1. Rename all the parameters, like we did for the variables
2. Add an extra argument to the list of parameters, representing the continuation. Let's call this parameter "k".
3. CPS transform the function body, and with the additional *context* that we need to *apply the function "k"* at some point.

Here's an example. First, let's use the cleaner Racket syntax to describe the transformation:

```
(lambda (x y) (+ x y)) ==CPS==> (lambda (cps_x cps_y k) (k (+ cps_x cps_y)))
```

For this example, we

1. Rename the parameters `x` and `y` to `cps_x` and `cps_y`
2. We add an extra parameter `k`
3. We CPS the body `(+ x y)` and obtain `(+ cps_x cps_y)`, but *afterwards* we need to call `k` to obtain `(k (+ cps_x cps_y))`

Here is the example in Haskell:

```
Prelude> cpsExpr (Lambda ["x", "y"] (Plus (Var "x") (Var "y"))) "" id
Lambda ["cps_x","cps_y","k"] (App (Var "k") [Plus (Var "cps_x") (Var "cps_y")])
```

Caveat

Note that figuring out when to apply the function "k" is a part of the problem. We cannot simply apply "k" to the body of the original functions (in direct style). That is, the following CPS transformation is incorrect:

```
(lambda (x y) (f x)) ==CPS==> (lambda (cps_x cps_y k) (k (cps_f cps_x))) ; INCORRECT!
```

Instead, we need to propagate the "k" inwards. If we follow the instructions for CPS transforming function applications in the next section, we should get a CPS transformed result like this:

```
(lambda (x y) (f x)) ==CPS==> (lambda (cps_x cps_y k) (cps_f cps_x (lambda (r) (k r))))
```

CPS Transforming Function Application

Function applications are the second most challenging portion of this assignment. First, let's look at the case where the function application happens *inside* another expression:

```
(+ 1 (f (+ x y) y))
```

Although the `(+ 1 _)` happens first, we'll focus on the CPS transformation of the function call `(f (+ x y) y)`. That we need to apply `(+ 1 _)` *afterwards* will be a part of the *context function*.

Here's what we need to do:

1. CPS transform the function subexpression: in our case the `f`
2. *after that is done* and we get the expression `cps_f`, we CPS transform the first argument subexpression (+ x y)
3. *after that is done* and we get the expression `(+ cps_x cps_y)`, we CPS transform the second argument subexpression `y`
4. *after that is done* and we get the expression `cps_y`, we have the CPS-ed version of all the different elements of our function call, and we can start to put it together. However, You might recall that `cps_f` takes *one extra argument*, representing the continuation. We need to supply that extra argument when we put together the function call.
5. The last argument to `cps_f` is the function's (`cps_f`'s) continuation, which will be a function. So, we need to *create a new lambda* that takes the result of the function, and ... what do we do with it? In the case of our example, we would like to apply the `(+ 1 _)`, which is work to be done *after* our function call is completed. That's exactly what needs to go into the body of our lambda!

The result of CPS transforming the above expression is this new one:

```
(+ 1 (f (+ x y) y))

== CPS ==>

(cps_f (+ cps_x cps_y) cps_y (lambda (result) (+ 1 result)))
```

In Haskell, we should have something similar to this (in your implementation, "result" may be called something else):

```
Prelude> cpsExpr (Plus (Literal $ Num 1) (App (Var "f") [Plus (Var "x") (Var "y"), Var "y"])) "" id
App (Var "cps_f") [Plus (Var "cps_x") (Var "cps_y"),
                   Var "cps_y",
                   Lambda ["result"] (Plus (Literal (Num 1)) (Var "result"))]
```

Let's trace through what happens in this above call:

The first thing we need to CPS transform is the outside `Plus`. To do this, we first CPS transform the left argument of the `Plus`, which is a number. Then after that, CPS the right argument of the `Plus`. This recursive call to `cpsExpr` will look something like this (but not exactly this):

```
cpsExpr (App (Var "f") [Plus (Var "x") (Var "y"), Var "y"])
        "some-value"
        (\result2 -> id $ (Plus (Literal $ Num 1) result2))
```

(In reality, `id` and `Literal $ Num 1` will probably be variables that are bound to those values.)

Now, when we make this recursive call, a few things will happen

1. CPS transform the function subexpression. So we call `cps_f` with the expression `(Var "f")`. The last argument of this recursive call is a function that does everything from steps 2-5.
2. CPS transform the first argument subexpression (the inside Plus). The last argument of this recursive call to `cps_f` will do everything from steps 3-5.
3. CPS transform the second argument subexpression. The last argument of this recursive call to `cps_f` will do everything from steps 4-5.
4. We put together the function call and create a new `App` with the new function expression and argument expressions. This new `App` takes one extra argument, which is the continuation of the function `cps_f`. This function will be...
5. ... created from this (Haskell) function from earlier: `\result2 -> id $ (Plus (Literal $ Num 1) result2)`! We need to create a new `Lambda` that looks something like this to be the final argument: `Lambda ["result"] (Plus (Literal (Num 1) (Var "result")))`

(Note that in your implementation, "result" may be called something else.)

Another Example

Let's look at a variation of the same example, where the function call happens at the top level.

```
(f (+ x y) y)

== CPS ==>

(cps_f (+ cps_x cps_y) cps_y (lambda (result) result))
```

Steps 1-4 is actually the same as above. The only difference is that there is no work to be done after the function. In this case, the *context function* should be the identity function, which is how we got the CPS transformed version above.

Notice though that we cheated a little here: in both cases, we named the variable of the lambda "result". This is problematic, because if we have multiple function calls as different levels, this variable could be shadowed! Your CPS transformation function should choose a unique enough name to avoid shadowing.

The *prefix string* argument to `cpsExpr` is meant to help you do exactly that. The idea is to update the *prefix string* whenever you recursively call `cpsExpr`, annotating the path that you are taking (e.g. left of a `+`, body of an `lambda`). You can decide on a variable naming scheme that aovid shadowing.

Note Avoiding shadowing is a little annoying, so we recommend handling the naming problem at the end.

Note: Iterating through each of the argument subexpressions is challenging. If necessary, you can assume that we will test with functions with at most 5 arguments (excluding the continuation argument). You can always come back and clean up your code later.

CPS Transforming If Expressions

If expressions are a little tricky, because only one of its two branches should be evaluated. Consider this expression:

```
(if (f x) (g x) (h x))
```

Here is how *not* to CPS transform this example:

```
(cps_f cps_x (lambda (result1)  ; this is wrong!
    (cps_g cps_x (lambda (result2)
        (cps_h cps_x (lambda (result3)
            (if result1 result2 result3)))))))
```

The problem with the above code is that **both functions g and h are called**. However, depending on the outcome of `(f x)`, we only want one of those two branches to be called.

```
(if (f x) (g x) (h x))

== CPS ==>

(cps_f cps_x (lambda (result1)
    (if result1
        (cps_g cps_x (lambda (result2) result2))
        (cps_h cps_x (lambda (result3) result3)))))
```

To make this exercise interesting, we won't spoil the answers for how to CPS transform if expressions.

As always, **start simple**. We do recommend leaving this part to the very end. Start by CPS transforming a number, then a plus/times/equal, then defining and evaluating simple functions before moving to more complex functions like "fac" (with a single recursive call) and finally "fib" (with multiple recursive calls).